WHAT IS CLAIMED IS:

1. A method of providing storage reclamation in a multiprocessor computer system, the method comprising:

maintaining respective reference counts for shared objects;

accessing pointers to the shared objects using lock-free pointer operations to coordinate modification of respective reference counts;

freeing storage associated with a particular one of the shared objects only once the corresponding reference count indicates that the particular shared object is unreferenced.

2. The method of claim 1, wherein the lock-free pointer operations ensure that:

if a number of pointers referencing the particular shared object is non-zero, then so too is the corresponding reference count; and

if no pointers reference the particular shared object, then the corresponding reference count eventually becomes zero.

3. The method of claim 2,

wherein at any given instant, a number of pointers to the particular shared object may differ from the corresponding reference count.

4. The method of claim 1, wherein the lock-free pointer operations include a load operation that loads a shared pointer value to a local pointer variable and employs:

a double-compare-and-swap (DCAS) primitive to increment a reference count of a first shared object, if any, referenced by the shared pointer value while ensuring continued existence thereof; and

a compare-and-swap (CAS) primitive to decrement a reference count of a second shared object, if any, referenced by a pre-load value of the local pointer variable.

5. The method of claim 1, wherein the pointer operations include a store operation that stores a local pointer value to a shared pointer variable and employs:

a compare-and-swap (CAS) primitive to increment a reference count of a first shared object, if any, referenced by the local pointer value;

a compare-and-swap (CAS) primitive to update the shared pointer variable with the local pointer value; and

a compare-and-swap (CAS) primitive to decrement a reference count of a second shared object, if any, referenced by a pre-store value of the shared pointer variable.

6. The method of claim 1, wherein the pointer operations include a copy operation that copies a local pointer value to a local pointer variable and employs:

a compare-and-swap (CAS) primitive to increment a reference count of a first shared object, if any, referenced by the local pointer value; and

a compare-and-swap (CAS) primitive to decrement a reference count of a second shared object, if any, referenced by a pre-copy value of the local pointer variable.

7. The method of claim 1, wherein the pointer operations include a destroy operation that:

decrements a reference count of a shared object identified by a supplied pointer value; and

frees the identified shared object if the corresponding reference count has reached zero.

8. The method of claim 7,

wherein, prior to the freeing, the destroy operation recursively follows pointers defined in the shared object if the corresponding reference count has reached zero.

9. The method of claim 1, employed in access operations on a composite shared object that includes zero or more of the shared objects.

10. The method of claim 9,

wherein the composite shared object is embodied as a double ended queue (deque);

wherein the shared objects include nodes of the deque; and

wherein the access operations implement push and pop accesses at opposing ends of the deque.

11. The method of claim 10, wherein the push accesses employ:

a pair of compare-and-swap (CAS) primitives to increment a reference count of a pushed node;

a double compare-and-swap (DCAS) primitive to splice the pushed node onto the deque while mediating competing accesses to the deque;

a pair of compare-and-swap (CAS) primitives to decrement the reference count of respective shared objects, if any, referenced by overwritten pre-splice pointer values.

12. The method of claim 11, wherein the pairs of compare-and-swap (CAS) primitives and the double compare-and-swap (DCAS) primitive are all encapsulated within one or more functions that implement an LFRCDCAS pointer operation.

13. A lock-free implementation of a concurrent shared object comprising:

plural component shared objects encoded in dynamically-allocated shared storage; and

access operations that, prior to attempting creation or replication of a pointer to any of the component shared objects, increment a corresponding reference count, and upon failure of the attempt, thereafter decrement the corresponding reference count,

the access operations decrementing a particular reference count, except when handling a pointer creation failure, no earlier than upon destruction of a pointer to a corresponding one of the component shared objects.

14. The lock-free implementation of a concurrent shared object as recited in claim 13, wherein the access operations employ lock-free, reference-count-maintaining pointer operations.

15. The lock-free implementation of a concurrent shared object as recited in claim 13, wherein the access operations include one or more of:

a lock-free, reference-count-maintaining load operation;

a lock-free, reference-count-maintaining store operation;

a lock-free, reference-count-maintaining copy operation;

a lock-free, reference-count-maintaining destroy operation;

a lock-free, reference-count-maintaining compare-and-swap (CAS) operation; and

a lock-free, reference-count-maintaining double compare-and-swap (DCAS) operation.

16. The lock-free implementation of a concurrent shared object as recited in claim 13, wherein each of the access operations are lock-free.

17. The lock-free implementation of a concurrent shared object as recited in claim 13, wherein the access operations employ either or both of a compare-and-swap (CAS) primitive and a double compare-and-swap (DCAS) primitive.

18. The lock-free implementation of a concurrent shared object as recited in claim 13, wherein the access operations employ emulations of either or both of the compare-and-swap and double-compare-and-swap operations.

19. The lock-free implementation of a concurrent shared object as recited in claim 18, wherein the emulation is based on one of:

a load-linked/store-conditional operation pair; and

transactional memory.

20. The lock-free implementation of a concurrent shared object as recited in claim 13,

wherein the incrementing and decrementing are performed using a synchronization primitive.

21. The lock-free implementation of a concurrent shared object as recited in claim 13,

wherein the concurrent shared object includes a doubly-linked list; and

wherein the access operations are performed using a synchronization primitive to mediate concurrent execution thereof.

22. A method of transforming an implementation of a concurrent shared data structure from garbage collection- (GC-) dependent to GC-independent form, the method comprising:

associating a reference count with each shared object instance;

modifying the implementation, if necessary, to ensure cycle-free garbage;

replacing pointer accesses in the implementation with corresponding lock-free, reference-count-maintaining counterpart operations; and

explicitly managing local pointer variables using a lock-free, reference-count-maintaining destroy operation that frees storage if a corresponding reference count has reached zero.

23. The method of claim 22, wherein the replacement of pointer accesses includes one or more of:

replacing an access that assigns a shared pointer value to a local pointer variable with a lock-free, reference-count-maintaining load operation;

replacing an access that assigns a local pointer value to a shared pointer variable with a lock-free, reference-count-maintaining store operation; and

replacing an access that assigns a local pointer value to a local pointer variable with a lock-free, reference-count-maintaining copy operation.

24. The method of claim 23, wherein the replacement of pointer accesses further includes:

replacing an access that assigns a shared pointer value to a shared pointer variable with:

a lock-free, reference-count-maintaining load operation to a local temporary variable;

a lock-free, reference-count-maintaining store operation from the local temporary variable; and

a lock-free, reference-count-maintaining destroy operation that frees storage associated with the local temporary variable if a corresponding reference count has reached zero.

25. The method of claim 22,

wherein the lock-free, reference-count-maintaining counterpart operations include object type specific instances thereof.

26. The method of claim 22,

wherein the lock-free, reference-count-maintaining counterpart operations are generic to plural object types.

27. The method of claim 22,

wherein the lock-free, reference-count-maintaining destroy operation is recursive.

28. The method of claim 22, further comprising:

generating a computer program product including a computer readable encoding of the concurrent shared data structure, which is instantiable in dynamically-allocated shared storage, the computer readable encoding further including functional sequences that facilitate access to the concurrent shared data structure and that include the lock-free, reference-count-maintaining counterpart operations.

29. A computer program product encoded in at least one computer readable medium, the computer program product comprising:

a representation of a shared object that is instantiable as zero or more component objects in dynamically-allocated shared storage of a multiprocessor;

at least one instruction sequence executable by respective processors of the multiprocessor, the at least one instruction sequence implementing at least one access operation on the shared object and employing one or more lock-free pointer operations to maintain reference counts for one or more accessed component objects thereof; and

the at least one instruction sequence further implementing explicit reclamation of the component objects, thereby freeing storage associated with a particular one of the component objects only once the corresponding reference count indicates that the particular component object is unreferenced.

30. The computer program product of claim 29,

wherein the zero or more component objects of the shared object are organized as a linked-list; and

wherein the at least one access operation supports concurrent access to the linked-list.

31. The computer program product of claim 29, at least partially implementing a mutator that provides explicit reclamation of the dynamically-allocated shared storage.

32. The computer program product of claim 29, at least partially implementing a garbage collector that reclaims shared storage dynamically-allocated for a mutator and, which employs the shared object in coordination thereof.

33. The computer program product of 29,

wherein the at least one computer readable medium is selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wire line, wireless or other communications medium.

34. An apparatus comprising:

plural processors;

one or more stores addressable by the plural processors;

one or more shared pointer variables accessible by each of the plural processors for referencing a shared object encoded in the one or more stores;

means for coordinating competing access to the shared object using one or more reference counts and pointer manipulations that employ one or more lock-free pointer operations to ensure that if the number of pointers to the shared object is non-zero, then so too is the corresponding reference count and further that if no pointers reference the shared object, then the corresponding reference count eventually becomes zero.

35. The apparatus of claim 34, further comprising:

means for freeing the shared object only once the corresponding reference count indicates that the shared object is unreferenced.